Data oddania sprawozdania: 09.11.2018 Termin zajęć: Piątek 17:05

Autor sprawozdania:
Jakub Majewski 238902

Prowadzący zajęcia:
dr inż. Zbigniew Buchalski

SPRAWOZDANIE

Projektowanie efektywnych algorytmów
Projekt 1

1. Wstęp

Należało zaprojektować własną implementację algorytmów brute force oraz branch and bound rozwiązujących problem asymetryczny komiwojażera.

Algorytm brute force opiera się na przeanalizowaniu wszystkich możliwych tras i wybraniu tej najkrótszej. Jego złożoność wynosi tyle ile jest możliwych permutacji N wierzchołków, a więc O(n!).

Algorytm branch and bound opiera się na znajdowaniu rozwiązania pomiędzy dwoma granicwami, które wraz z kolejnymi iteracjami algorytmu zwężają się aż pomiędzy nimi będzie znajdowało się tylko najlepsze rozwiązanie (bądź rozwiązania, o ile jest więcej niż jedna prawidłowa ścieżka). Jedna z granic to najlepsze dotychczasowe rozwiązanie. Druga granica natomiast opisuje najlepsze drugie z kolei najlepsze możliwe rozwiązanie. Jeżeli okaże się, że wybranie dowolnego następnego kroku spowoduje przekroczenie wartości drugiej granicy to algorytm cofa się do stanu drugiej granicy i oznacza ją jako pierwszą, a poprzednie rozwiązanie jest blokowane, aby algorytm nie próbował znów sprawdzać danego odgałęzienia. Złożoność obliczeniowa w porównaniu do algorytmu brute force nie jest już tak prosta do określenia. W najgorszym przypadku algorytm branch and bound może mieć złożoność O(n!) gdyż będzie musiał przeanalizować wszystkie przypadki, w najlepszym przypadku natomiast algorytm natychmiast znajdzie najlepsze rozwiązanie przy przejściu przez N wierzchołków, a więc jego złożoność dla najlepszego przypadku będzie równa O(n). Wiadomo zatem, że średnia złożoność tego algorytmu będzie znajdowała się w przedziale (O(n), O(n!)).

Główna różnica pomiędzy algorytmem branch and bound, a brute force jest taka, że tego drugiego jest całkowicie zależna od wprowadzonych danych, natomiast algorytm brute force, zawsze musi przeanalizować n! przypadków, niezależnie od danych.

2. Przykład działania algorytmu branch and bound dla przykładowej instancji.

Macierz bazowa:

-	20	30	10	11
15	-	16	4	2
3	5	-	2	4
19	6	18	-	3
16	4	7	16	-

Redukcja macierzy oraz obliczanie minimalnego kosztu:

-	10	17	0	1	**10**
12	-	11	2	0	**2**
0	3	-	0	2	**2**
15	3	12	-	0	**3**
11	0	0	12	0	**4**
1	**0**	**3**	**0**	**0**	**Min**

Min = (10+2+2+3+4)+(1+0+3+0+0) = 25

Najlepsze możliwe teoretyczne rozwiązanie ma wartość równą 25(Min).

Startując od pierwszego wierzchołka możemy wyruszyć w jeden z 4 pozostałych. A więc aktualnie możemy wybrać się do wierzchołka 2,3,4 albo 5.

Obliczamy zatem dla każdego wierzchołka (2,3,4,5) najlepszy koszt po przejściu przez dany wierzchołek.

W poprzedniej macierzy dla każdego dostępnego wierzchołka usuwamy kolumny i wiersze oraz komórkę dla połączenia x->1, ponieważ będziemy wybierać za chwilę drogę 1->x. W ten sposób otrzymamy 4 różne macierze.

Redukujemy otrzymane macierze i obliczamy minimalne koszty korzystając ze wzoru:
Koszt = c(x,y) + v + ^v
gdzie:
c(x,y) - koszt przejścia z x do y (dla zredukowanej macierzy bazowej).
v - wartość minimalna (Min) otrzymana podczas redukcji macierzy
^v - poprzednia wartość minimalna (dla poprzedniego wybranego wierzchołka)

Przykład zredukowanej macierzy 1->3:

-	-	-	-	-	**0**
1	-	-	2	0	**0**
-	3	-	0	2	**0**
4	3	-	-	0	**0**
0	0	-	12	0	**0**
11	**0**	**0**	**0**	**0**	**Min**

Min = (11+0+0+0+0)+(0+0+0+0+0) = 11
Koszt = c(1,3)+v+^v = 17+11+25 = 53

Po obliczeniu wszystkich macierzy dla przejść: 1->2, 1->3, 1->4, 1->5 Wybieramy wierzchołek, w którym wyszedł najmniejszy koszt(Koszt).	W tym wypadku będzie to wierzchołek 4. Po przejściu do 4. wierzchołka mamy następujące dostępne drogi: 1->4->2, 1->4->3, 1->4->5. Obliczamy macierze przejść dla tych możliwych dróg, a następnie koszty i wybieramy drogę o najmniejszym znanym dotychczas koszcie. Algorytm powtarzamy, aż do otrzymania całej trasy.

3. Opis implementacji algorytmu branch and bound.
a) Obliczanie macierzy przejść i kosztów.
b) Wybór najlepszej wybierając tą o najlepszym koszcie.
c) W przypadku konieczności wycofania się, algorytm resetuje stan macierzy do stanu, w którym będzie mógł wybrać trasę o mniejszym koszcie oraz dodaje trasę prowadzącą do gorszych rozwiązań poprzez dodanie jej do listy tras zbanowanych.

W algorytmie wykorzystałem jedynie własne struktury danych (wyjątek stanowi std::pair, który jest zwykłą szablonową strukturą dwuelementową):
a) `MinHeap` - Sterta służąca jako kolejka priorytetowa dla wartości minimalnych
b) `UnorderedVector` - Własna implementacja `std::vector` z biblioteki standardowej C++, z tą różnicą, że usuwanie z niego elementów posiada złożoność czasową O(1) przez zastosowanie zwykłej jednostronnej zamiany elementu usuwanego na element będący na końcu wektora, w efekcie końcowym elementy w tym wektorze, po usunięciu dowolnej środkowej/początkowej komórki, nie zachowują swojej kolejności w jakiej zostały dodane.
c) `Matrix` - Zwykła implementacja dwuwymiarowej tablicy z możliwością odczytywania danych z pliku oraz kilkoma innymi udogodnieniami jak redukcja macierzy.

4. Plan eksperymentu.

Rozmiar użytych struktur danych: `6, 7, 8, 9, 10, 11, 12.`

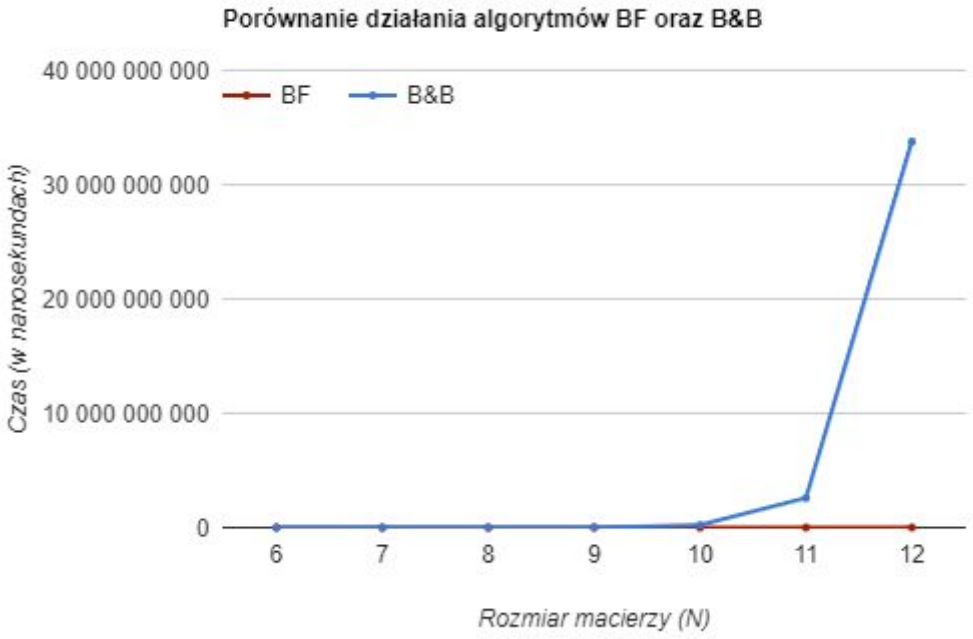
Sposób generowania danych: Za pomocą funkcji `std::rand()` z ustawianiem ziarna losowości za pomocą `std::srand(std::time(nullptr))`.

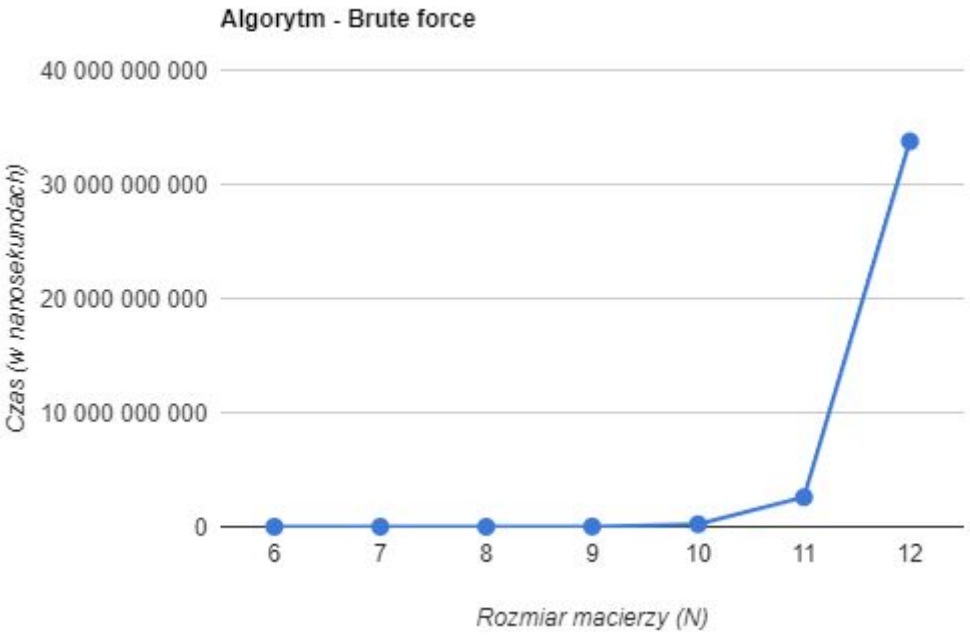
Metoda pomiaru czasu: Za pomocą funkcji `std::chrono::high_resolution_clock::now()` z biblioteki standardowej C++.

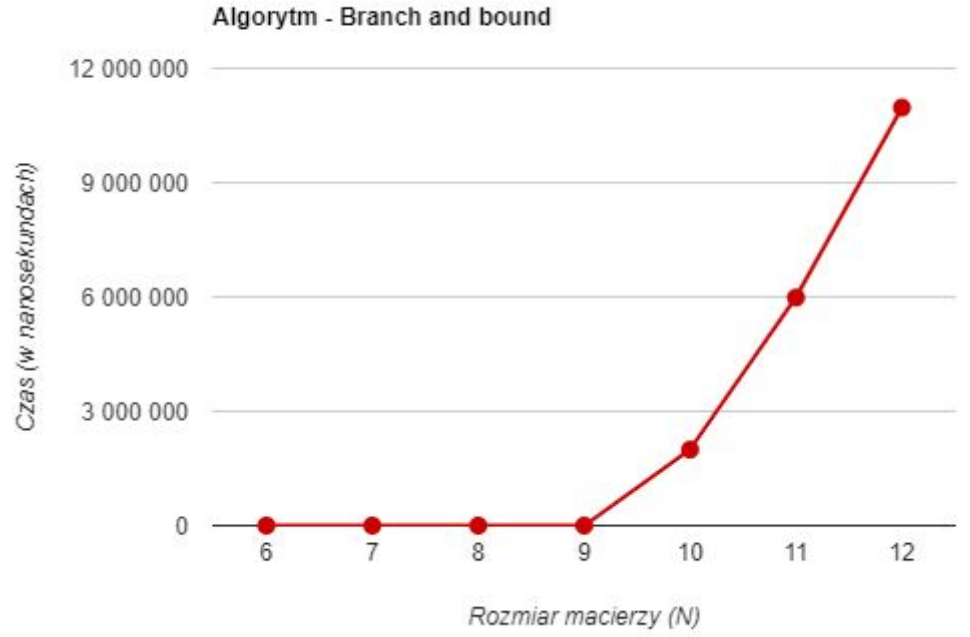
5. Wyniki eksperymentów:

Tabela wyników:

Wielkość macierzy (NxN)	Uśredniony czas wykonywania algorytmu	
	Brute force	Branch and bound
6x6	0	0
7x7	997500	0
8x8	1994300	0
9x9	19946600	0
10x10	223402500	1994000
11x11	2589115300	5985000
12x12	33767902800	10970700




Algorytm - Brute force
40 000 000 000
30 000 000 000
20 000 000 000
10 000 000 000
0
Czas (w nanosekundach)
6
7
8
9
10
11
12
Rozmiar macierzy (N)


Algorytm - Branch and bound
12 000 000
9 000 000
6 000 000
3 000 000
0
Czas (w nanosekundach)
6
7
8
9
10
11
12
Rozmiar macierzy (N)